General Store: Speculative Address Translation in x86 Processors

Yanik Kleibrink
Technical University of Darmstadt

Anirban Chakraborty
Max Planck Institute for Security and Privacy

Yuval Yarom
Ruhr University Bochum

Abstract

The Spectre family of attacks exploits speculative execution to access secret data and transmit it across isolation boundaries using a microarchitectural covert channel. Whereas prior work has predominantly examined the use of speculative loads for constructing such channels, we investigate speculative stores and flush operations across a wide range of Intel and AMD processors. We find that speculative memory operations either update the data cache or initiate page table walks. Depending on the microarchitecture, the walk may complete and update the TLB or be aborted after populating data caches, leaving clear microarchitectural traces of the translation. We further characterize the effects of page table attributes, memory fences, and cache-coherence states on this behavior. Building on these findings, we introduce a covert channel that leverages only the page table walk activity of speculative stores to encode information, without relying on store-induced cache fills. Finally, we demonstrate that Speculative Load Hardening (SLH)—a widely deployed Spectre-v1 mitigation in LLVM—does not prevent speculative store-based leakage of register values, consistent with its threat model and design assumptions.

1 Introduction

The discovery of Spectre [18] in 2018 revealed that optimization techniques in modern processors, such as *speculative execution*, that were hitherto considered benign performance improvements, can be exploited to leak sensitive information through microarchitectural side channels. Since then, a plethora of works [1, 2, 3, 4, 9, 10, 12, 15, 16, 17, 19, 23, 37] have explored various facets of speculative execution attacks that are collectively referred to as Spectre-class attacks [7]. Unlike other microarchitectural attacks, such as Meltdown [6, 20], Spectre-class attacks—specifically Spectre-v1 (Spectre-PHT)—do not rely on specific software or hardware vulnerabilities, but instead exploit the fundamental design features of modern processors. Consequently, Spectre-class attacks are considered harder to mitigate, as they often necessitate substantial changes to processor architecture and difficult to effectively address through microcode updates or software patches alone [18].

A Spectre attack exploits speculative execution to execute code outside the nominal execution of the program. In Spectre-v1 [18], speculation is the result of a mispredicted conditional branch. Other variants of Spectre exploit other speculation causes, such as indirect branches [18] or the return stack buffer [19, 23]. The mispredicted code accesses secret data, e.g., bypassing a bounds check [18] or through speculative type confusion [2, 17], and transmits it through a microarchitectural covert channel. Most Spectre attacks use the Flush+Reload channel [34]. However, other channels have also been demonstrated [4, 15, 24, 29, 38].

Due to the prevalence of load-based covert channels for Spectre-class attacks, the majority of defenses focus on restricting load instructions [38] or mitigating their microarchitectural effects [32, 35]. In contrast, the role of store instructions in Spectre remains comparatively under-explored. Dolma [22] observes that speculative stores can trigger page table walks and potentially leak information through the translation lookaside buffer (TLB). Patrigniani and Guarnieri [27] present a theoretical model in which speculative stores leak both the accessed address and the stored data, motivating store-protecting defenses such as USLH [38]. Although not used for a Spectre-class attack, Zhang et al. [36] show how to transmit information using transient writes in conjunction with the MWAIT and MONITOR instructions. Similarly, multiple works show that address translation can leak sensitive information through TLB state and page table walk side effects [5, 13, 14, 30, 39]. Beyond covert channels, Kiriansky and Waldspurger [16] study how speculative stores interact with younger speculative loads. However, despite this body of work, *the impact of address-translation effects speculatively triggered by store and cache flush instructions* has not, to the best of our knowledge, been examined.

Contribution. In this paper, we closely investigate the interactions between transient store and flush instructions and the CPU memory subsystem, across multiple microrachitectures. We expand and systematize prior work that already suggests that speculative store instructions may trigger page table walks [22] or observes that speculative stores can also bring the targeted cache line into the cache on some microarchitectures [25, 29]. We first map the behavior of the instructions under speculation across a wide variety of Intel and AMD processors. We examine multiple properties, including whether they initiate a page table walk, to what extent the page table walk proceeds, and the impact of fence instructions and page permissions on the address translation process.

Our experiments show that in contrast with some older microarchitectures, on many newer architectures speculative store instructions do bring the targeted cache line to the cache. Moreover, we find that while all tested instructions do start the page table walk speculatively on nearly all microarchitectures, the extent to which it proceeds varies significantly. Notably, on several Intel processors, the page table walk can be interrupted mid-way, preventing the translation from completing. In contrast, on AMD processors, the walk consistently proceeds to completion regardless of speculation.

Building on our observations, we demonstrate a covert channel that leverages speculative stores—not through direct cache updates, but through leakage induced by the address translation of speculative store targets. Finally, we evaluate the effectiveness of compiler-based mitigations such as SLH and USLH against store-based Spectre attacks and discuss potential directions for mitigation.

In summary, the contributions of this work are:

- We evaluate the behavior of speculative stores, loads, and flushes on different microarchitectures, revealing hitherto unknown behavior (Section 3).
- We present a covert channel that leverages the side-effects of the page table walk to encode information, while not relying on store-induced cache fills (Section 4).
- We evaluate compiler-based mitigations, such as SLH and USLH, against store-based Spectre attacks (Section 5).

Ethical Considerations and Open Science. Our work does not exploit any new vulnerabilities. Consequently, we did not engage in a formal vendor disclosure process. The source code for all experiments is available at `https://github.com/0xADE1A1DE/GenStore`.

2 Background

In this section, we review speculative execution, spectre attacks and the address translation process on modern x86 processors.

2.1 Speculative and Out-of-Order Execution

Modern processors employ deep pipelines and complex microarchitectures to improve throughput and runtime performance. These pipelines are generally divided into two stages: a frontend, responsible for fetching and decoding instructions, and a backend, that executes them and performs memory operations. To exploit instruction-level parallelism, the frontend can fetch and decode multiple instructions simultaneously, while the backend may execute instructions out-of-order to maximize utilization of execution units and other resources. Consequently, the order in which instructions execute may differ from the program order specified by the programmer. To ensure correct program semantics, the results of out-of-order execution are committed in program order.

In many cases, the frontend cannot determine the outcome of a branch until older instructions execute, which may take many cycles. To avoid pipeline stalls, the processor predicts branch outcomes and speculatively executes instructions along the predicted path. If the prediction is correct, execution proceeds without interruption, yielding a performance benefit. Conversely, in the case of an incorrect prediction, the speculatively executed instructions are squashed, and execution resumes along the correct path. Importantly, even when rolled back, speculatively executed instructions can leave microarchitectural side effects, including cache or buffer state changes, enabling attacks such as Spectre.

2.2 Spectre Attacks

A natural consequence of out-of-order and speculative execution is that instructions may execute transiently, even if they would not appear in the nominal program order. Although these instructions are eventually squashed, their microarchitectural side effects—such as cache fills, TLB updates, or predictor state changes—can persist and be observed through side channels. The Spectre family of attacks [18] demonstrates how such transient execution can leak sensitive information.

In this paper, we focus on *Spectre-v1 (Spectre-PHT)*, which leverages speculative loads to access protected memory. In this variant, an attacker mistrains a conditional branch so that a subsequent load instruction speculatively reads from a memory location that would be inaccessible under the correct program flow. Listing 1 shows a minimal Spectre-v1 gadget. By controlling `index`, the attacker can speculatively access an out-of-bounds element of `array1`, and encode the value of `array1[index]` into the cache state of `array2`. This state can then be probed using a cache-based side channel, such as Flush+Reload [34], to recover the secret.

```
if (index < array1_size) {
  x = array2[array1[index] * 4096];
}
```

Listing 1: A simple Spectre-v1 gadget that leaks the value of `array1[index]` through a cache-based side channel.

Defenses against Spectre can be broadly classified into three categories: hardware changes, speculation barriers, and compiler-based mitigations. In this work, we focus on the latter two. Speculation barriers, such as the LFENCE instruction on x86 processors, prevent execution of instructions following the barrier until all instructions before it commit. Inserting a speculation barrier at each possible outcome of conditional branches prevents the attack, albeit at a significant performance cost [18]. Furthermore, it is currently unknown whether all speculation barriers are effective against all forms of speculation—for example, some barriers may prevent speculative instruction execution but may still allow speculative address translation.

Compiler-based countermeasures, such as Speculative Load Hardening (SLH) [8], protect against leaks by tracking the speculation state and masking the results of loads on mis-speculated paths. However, previous works [38] have shown that SLH may be bypassed in certain scenarios, particularly when combined with other microarchitectural features [32, 35]. Furthermore, SLH primarily focuses on loads, leaving the behavior of stores under speculation less explored.

2.3 Address Translation in x86 Processors

Modern processors use virtual memory to provide each process with an isolated address space. When a memory instruction references a virtual address, the processor must translate it to a physical address using a multi-level page table hierarchy managed by the operating system. This translation is performed by the Memory Management Unit (MMU) through a series of *page table* lookups.

The virtual address is divided into multiple fields to index into the page tables. Modern x86 processors use a hierarchical structure with four or five levels for page tables. As all of our test machines use a four-level hierarchy, our description is limited to that case. For 4 KB pages, the 16 most significant bits of the virtual address are ignored, the next 36 bits are split into four 9-bit sections that index into the four page table levels, and the 12 least significant bits specify the offset within the target page. Each page table entry (PTE) contains the physical address of the next-level page table along with permission and status bits. The final-level PTE provides the physical address of the target page.

Figure 1 illustrates the address translation process on modern x86 processors. To translate a virtual address, the MMU performs a *page table walk (PTW)*, starting from the root of the hierarchy (the PGD) and using the relevant bits of the virtual address to index sequentially into each level. To reduce the overhead of consulting four (or five) translation tables for each memory access, modern processors cache complete address translations in a dedicated cache, called Translation Lookaside Buffer (TLB), and intermediate higher-level page table entries in page walker caches, also known as *page structure caches* in the Intel nomenclature.

Crucially for speculative execution, speculative loads and stores may trigger page table walks even if the instructions are eventually squashed. On some microarchitectures, speculative accesses can bring intermediate or final PTEs into the cache or TLB, producing observable microarchitectural side effects. Speculative stores, in particular, may not update architectural state but can still cause cache footprints via address translation, forming the basis for the store-based side channels analyzed in this work.

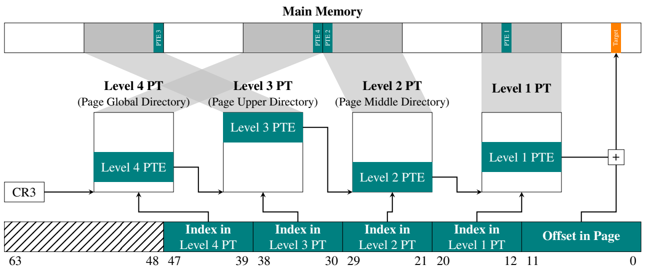

Figure 1: The address translation process on modern Intel and AMD Processors. PTE is used as an abbreviation of page table entry, and PT is an abbreviation of page table.

3 Analysis of Speculative Store Operations

To understand how transient stores interact with the memory hierarchy, particularly whether they trigger address translations and how far the page table walk progresses, we now analyze the behavior of speculative `store` instructions across a wide range of Intel and AMD processors. We further examine the effects of fence instructions on speculative stores and the influence of various page permission bits. In particular, we seek to answer the following research questions:

RQ1: Do speculative stores trigger address translations, and if so, how far does the page table walk progress?

RQ2: How do fence instructions affect the address translation process of speculative stores?

RQ3: How do different page permission bits influence the behavior of speculative stores?

To address these questions, we employ a two-part experimental setup. First, we design a *speculation gadget* based on return stack buffer (RSB) speculation to reliably trigger speculative stores. This approach allows us to create a controlled transient execution window without the need for complex branch predictor training. Second, we develop a *custom Linux kernel module* that provides fine-grained control over the page table layout in memory. This module enables us to manipulate page table entries and permissions, facilitating detailed analysis of the address translation process during speculative store operations.[1]

[1]Parts of the functionality are similar to https://github.com/misc0110/PTEditor.

```
1  call level_a
2  jmp end
3  level_a:
4      call mispredict_return
5      # Code executed speculatively (Here a memory
        ↪ instruction)
6      lfence # Prevent the execution from speculatively
        ↪ continuing past this point.
7  mispredict_return:
8      pop %rdi # Pop the top of the return stack
9      clflush (%rsp)
10     cpuid
11     ret # Speculatively returns to line 5
12 end:
```

Listing 2: Code for triggering transient execution via return-stack speculation.

3.1 Speculation Gadget Design

The original Spectre-v1 attack [18] uses conditional branches to trigger speculative loads (cf. Section 2.2). However, training conditional branches can be noisy and slow, leading to inconsistent speculation windows. To achieve more reliable control, we instead employ Return Stack Buffer (RSB)-based speculation [19, 23, 31] to trigger speculative store instructions. The RSB is a hardware structure that stores the return addresses of recently executed CALL instructions. When a return instruction is next encountered, the processor predicts its target by popping the top entry from the RSB. If the RSB no longer syncs with the architectural call stack—for instance, due to deliberate modifications of the stack pointer—the processor may speculatively return to an incorrect address, causing transient execution along a mispredicted path.

Listing 2 shows our speculation gadget, which consists of two nested calls, first to `level_a` and then to `mispredict_return`. These calls push return addresses both onto the software stack and into the RSB. Now, we manually modify the top of the stack (line 8), thereby desynchronizing the stack and the RSB. Next, we flush the cache line containing the top of the return stack (line 9) to delay the resolution of the true return address. When the subsequent RET (line 11) is encountered, the processor cannot immediately retrieve the correct return address from memory and therefore speculatively returns to the stale entry still present in the RSB, i.e., the instruction following the call to `mispredict_return` (line 5). Finally, to detect the presence of cache lines in the data caches, we measure the time to access them using the RDTSCP or RDPRU instructions.

3.2 Kernel Module for Page Table Control

Our evaluation requires fine-grained control over page table state, which is not accessible from user-level program in Linux due to security restrictions. To enable this, we develop a custom Linux kernel module that allows us to allocate pages, set up multi-level page tables, control the state of the data cache, TLB, and page walker caches (PWC), and modify page table entry (PTE) attributes.

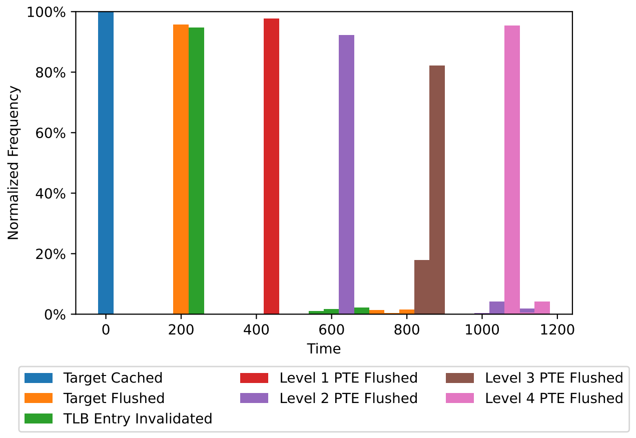

Figure 2: Time to load a target memory address with subsets of its page table entries (PTE) flushed. Here *i* PTE flushed means that all lower level PTEs are flushed as well.

At initialization, the kernel module allocates a page that serves as the target of the speculative memory operation. Initially, the target page shares higher-level page table entries with the stack (`rsp`), which prevents independent cache control of the corresponding PTEs.

To avoid any overlap between the page table walks of the `rsp` and the speculative target, we remap the target's physical page to a virtual address that differs from the `rsp` in bits [47:42]. This ensures that the PTEs of the two addresses reside in distinct cache lines, preventing a page table walk for one address from inadvertently loading PTEs of the other.

A page table walker routine exposes the (virtual) addresses of the PTEs corresponding to the target page. To validate these mappings, we ① flush the PTEs from the TLB and PWCs using the INVLPG instruction, ② systematically flush higher-level PTEs from the memory caches, and ③ measure the resulting access latency to the target page. The invalidation of the TLB and PWCs forces the processor to walk the page table, and thus load all PTEs, to resolve the target address. Each additional page table level flushed from the memory caches incurs an additional load from DRAM and thus results in a measurable increase in the access latency (cf. Figure 2). This delay confirms that the flushed entries correspond to valid PTEs in the translation hierarchy.

3.3 Experimental Setup and Evaluation

We now describe the experimental setup and our observations regarding the behavior of speculative store operations across

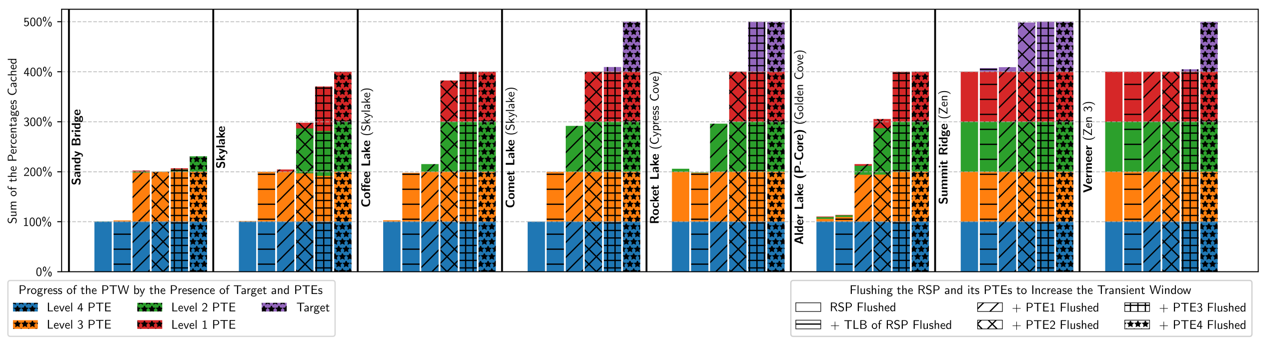

Figure 3: Progress of the PTW of a speculative store to a target memory address.

various processors.

Platform Configuration. We conduct our experiments on a diverse set of Intel and AMD processors, spanning multiple microarchitectures and generations. (See Table 3 in the Appendix for full details.) All machines run Linux with kernel versions ranging from 5.5 to 6.17, with IBT disabled, as required for our kernel module. In each machine, all cores except core 0 and the core that executes the experiment are disabled. To reduce noise due to timing variations, we disable frequency scaling by setting the `scaling_governor` to `performance`. To ensure the largest possible consistency across the different operating system environments, we statically compile all of our tests with musl[2] using GCC 10 and distribute those static executables to all of the test machines.

Methodology. We use the RSB speculation gadget (cf. Section 3.1) to induce speculative memory operations (e.g., a `store`) to a chosen target address and manipulate the corresponding page table entries via our custom kernel module (cf. Section 3.2). To characterize how page table walks, fence instructions, and page-permission bits affect speculative stores, we run a suite of controlled experiments. Each test is configured from user space via an `ioctl` call that conveys the memory-operation type and the desired microarchitectural state (TLB, PWC, and data caches). A second `ioctl` supplies a user-space buffer into which the kernel writes the observed timing-frequency histogram, enabling efficient collection of large sample sets.

Execution proceeds in five stages. First, the microarchitectural state is prepared by selectively loading or flushing the target and related page table entries to achieve the desired baseline configuration. Second, the specified page table entry attributes (e.g., present, accessed, user, writable), that generate exceptions, are modified as side-effects of other instructions, or are architecturally incorrect, are adjusted to reflect the desired configuration. These control the behavior of memory accesses during the transient execution window. Third, the RSB speculation gadget that performs speculative store operations to the target address is triggered. Fourth, any modified PTE attributes are restored to their original state to ensure architectural correctness. Finally, the latency of accessing the target or its page table entries is measured to observe the effects of speculative stores on the memory hierarchy. Furthermore, to mitigate system noise and ensure statistical significance, each test is repeated for a minimum duration and iteration count. We defer a more detailed explanation of the test phases to the subsequent sections, where we discuss specific experiments.

Results and Observations. To validate our premise that address translation, and the associated PTW it can trigger, lead to a meaningful and detectable trace in the caches, we utilize our custom Linux kernel module to flush a target from all caches and invalidate its entries in the TLB / PWC. Afterwards, the load of target is timed while iteratively flushing higher PTEs. The results are presented in Figure 3. As expected the latency of the load of target increases by about 1 DRAM access latency for each additional PTE flushed from the caches.

Using our custom Linux kernel module, we next study the PTW started by a speculative memory operation by precisely observing which PTEs are loaded by the table walk as it progresses (cf. Figure 3 or Appendix, Table 4). To analyze the effects of the length of the transient window, it is slowly increased. On AMD Zen processors, we observe that a PTW is never stopped or interrupted once started. On Intel the PTW can be interrupted on all generations from Sandy Bridge to Alder Lake. In particular, increasing the length of the transient window allows the PTW to progress further.

Table 1 shows a concise summary of our results. For each configuration, we examine whether a PTW is started (○), whether the TLB is also updated (◐), and whether the target is cached (●). Crucially we observe that in nearly all cases all memory operations (e.g., CLFLUSH) that require virtual address translation can speculatively start this address transla-

[2] `https://musl.libc.org/`

Table 1: Effects of speculative execution on the address translation process. A cross (✕) indicates that the address translation process does not start. An empty circle (○) means that the PTW starts but the TLB does not update. In the case of a half circle (◐), the TLB is updated and a full circle (●) indicates that the target line is cached in the D-caches. The abbreviations NA = not accessed and RO = read-only are used.
* indicates that the test runs in user mode, with the only possible results being cached (●) and uncached (✕).
† indicates that TLB updates are not checked for this test.

		Loads										Stores										Flushes						
		Regular	SFENCE	MFENCE	LFENCE	Exclusive*	Modified*	Shared*	RO Page	Clean Page	NA Page†	Regular	SFENCE	MFENCE	LFENCE	Exclusive*	Modified*	Shared*	RO Page	Clean Page	NA Page†	Regular	SFENCE	MFENCE	LFENCE	RO Page	Clean Page	NA Page†
Sandy Bridge	(Sandy Bridge)	●	●	✕	✕	●	●		●	●	○	◐	◐	✕	✕	●	●		◐	◐	○	◐	◐	✕	✕	◐	◐	○
Skylake	(Skylake)	●	●	✕	✕	●	●		●	●	○	◐	◐	✕	✕	●	●		◐	◐	○	◐	◐	✕	✕	◐	◐	○
Coffee Lake	(Skylake)	●	●	✕	✕	●	●	●	●	●	○	◐	◐	✕	✕	●	●	●	◐	◐	○	◐	◐	✕	✕	◐	◐	○
Comet Lake	(Skylake)	●	●	✕	✕	●	●	●	●	●	○	●	●	✕	✕	●	●	●	◐	◐	○	◐	◐	✕	✕	◐	◐	○
Rocket Lake	(Cypress Cove)	●	●	✕	✕	●	●	●	●	●	○	●	●	✕	✕	●	●	●	◐	◐	○	◐	◐	✕	✕	◐	◐	○
Alder Lake (P-Core)	(Golden Cove)	●	●	✕	✕	●	●	●	●	●	○	◐	◐	✕	✕	●	●	●	◐	◐	○	◐	◐	✕	✕	◐	◐	○
Alder Lake (E-Core)	(Gracemont)	✕	✕	✕	✕	●	●	●	✕	✕	✕	✕	✕	✕	✕	●	●	●	✕	✕	✕	✕	✕	✕	✕	✕	✕	✕
Summit Ridge	(Zen)	●	●	✕	✕	●	●	●	●	●	○	●	●	✕	✕	●	●	●	○	○	○	○	○	✕	✕	○	○	○
Vermeer	(Zen 3)	●	●	✕	✕	●	●	●	●	●	○	●	●	✕	✕	●	●	●	◐	○	○	◐	◐	✕	✕	◐	◐	○

tion despite the memory operation itself never being speculative. This can only be prevented by inserting an LFENCE or SFENCE before the memory operation. Interestingly, the behavior concerning speculative stores has changed frequently between the microarchitectures with newer Intel architectures all exhibiting some type of caching for speculative stores and older architectures usually only updating the TLB but not caching the memory operand. On all AMD Zen processors the target of the speculative store is cached.

Observation 1: Instructions with a memory operand (e.g., CLFLUSH), whose effect (e.g., flushing the memory operand's cache line) is never realized in a transient context, can still trigger address translation of the operand inside the transient window.

Observation 2: On newer architectures the targets of speculative stores and loads can be cached irrespective of the coherency state.

3.4 Page Table Walk of Speculative Memory Instructions

We now analyse the address translation process of speculative memory operations (`load`, `store` and `flush`) to understand how far the page table walk progresses and whether the TLB is updated.

To this end, we first study whether speculative memory operations lead to the caching of their target memory operand. The operand is first flushed from all data caches and then the RSB gadget is used to trigger the speculative execution of the memory operation. The time to access the memory operand afterwards reveals the caching effect and the progress of the page translation process. As shown in the previous subsection (cf. Figure 2), the range of access time can be used to distinguish between different cache and TLB states.

We first observe that, with the exception of Gracemont, speculative loads always bring their target into the data caches. This is shown under the *Regular* column for `load` in Table 1 where all other tested processors show a full circle (●) indicating that the target was cached. Speculative stores also bring their target into the data caches (● in `store` under *Regular* column) on all AMD Zen architectures and on Intel microarchitectures starting from 8th Generation (Coffee Lake). However, on older Intel microarchitectures (e.g., Sandy Bridge and Skylake), speculative stores do not bring their target into the data caches (◐ in `store` under *Regular* column). Furthermore, we observe that speculative `clflush` operations never bring their target into the data caches (✕ in `flush` under *Regular* column) across all tested processors. Note that on Gracemont, ✕ means that the target is not cached. This observation only pertains to kernel-space executions, in user-space speculative stores are also cached.

As we observe that certain processor families do not bring the target of speculative stores and flushes into the data caches, we next investigate how far the page table walk progresses while translating the target virtual address of a speculative store. In particular, we monitor the loading of the individual PTE entries from different levels of the page table hierarchy into the data caches by the PTW. While the activation of the PTW can be detected by the delay it causes to the associated memory operation (cf. Figure 2), directly timing the loads of the individual PTEs allows for a fine-grained observation of when which PTE was loaded.

It is important to note that the RSB gadget creates a transient window that lasts until the load that resolves `rsp` completes. The duration of this window therefore bounds how far a page table walk (PTW) can progress. Consequently, a speculative address translation can produce two distinct outcomes: (1) the PTW completes and the translation is available in the TLB, or (2) the PTW is aborted before it reaches the page table level being monitored.

To test whether speculative address translations populate the TLB, we first trigger a speculative memory access whose translation is expected to complete within the transient window. Next, we flush all page table cache lines from the data caches while explicitly preserving the corresponding TLB entry, and evict the target line itself. Finally, we issue an architectural load to the same virtual address. If this load completes without invoking a new page table walk, we conclude that the translation had been installed in the TLB during speculation. The triggering of a PTW can be detected either by measuring the load latency (cf. Figure 2) or by directly timing accesses to the level 1 PTE. We observe that, with the exception of Summit Ridge, the TLB is always updated when a PTW is started. In particular, this pertains to all examined Intel processors.

Similarly, to provoke PTW aborts we rely on the RSB gadget to create a transient execution window whose length depends on the latency of resolving the return address (`rsp`). We extend this window by flushing the PTEs that map `rsp` so that their translation requires main-memory accesses.[3] Before each trial we establish a clean baseline by flushing the target, invalidating its TLB and PWC entries, and evicting all candidate PTE cache lines. We then execute the RSB gadget and, immediately after speculation, time loads to each PTE (and to the target) to determine which entries the PTW fetched. Figure 3 shows the result for all processor generations.[4] Note that the PTW completes even for the relatively short transient window created by only flushing the `rsp` on Summit Ridge. This leads us to conclude that PTWs on AMD processors are never interrupted.

By tuning the transient-window length in this way, we can control the PTW's abortion point so that only PTEs above a chosen level are fetched. This enables a practical framework to observe the PTW progression and can be used to construct a page-walk side-channel order oracle [30]. With the window set to approximately a single memory-access duration, repeated trials permit the PTW to fetch PTEs incrementally from the highest level down.

[3]Alternatively, it is possible to introduce a data dependency that prevents `rsp` from being resolved until other memory operations complete. Regardless, the remapping of target remains necessary to avoid sharing higher-level PTEs with other pages.

[4]On Gracemont, we could not observe any caching of PTEs usually associated with a PTW, see also Table 4 in the Appendix.

Observation 3: On nearly all tested processors, speculative memory operations trigger page table walks, and the address translation is cached in the TLB. Moreover, these page table walk might always be interrupted.

3.5 Effect of Fencing on Address Translation

Since the inception of speculative execution, it has been well understood that certain fence instructions can prevent speculation across instruction boundaries [7, 18]. However, the precise impact of these fences on the address translation of subsequent memory operations remains unexplored. In the previous subsection, we showed that speculative memory operations can trigger page table walks and update the TLB. We now investigate whether fence instructions can prevent this.

For these experiments, we first flush the top entry of the RSB and all its corresponding page table entries from every cache level. Next, we flush the target address, its entries in the TLB and page-walk caches, and its level-1 PTE. From the results of the previous section, we know that this transient window is sufficiently long for a full page table walk and the caching of the target (for both `load` and `store` operations) to complete. We then insert the fence instructions under test (LFENCE, MFENCE and SFENCE) immediately before the memory instruction in line 5 of Listing 2. This placement allows us to directly observe whether the fence delays or suppresses the initiation of the page table walk and to analyze its effect on the caching behavior of the memory instruction.

Figure 4 shows the results of this experiment for the Intel Core i7-8700 (cf. also Table 6 in the Appendix). The `lfence` and `mfence` instructions completely prevent the address translation and caching of the target of a transient load and store. An `sfence` instruction permits the address translation to complete and the target to be loaded.

Observation 4: On all tested microarchitectures, MFENCE and LFENCE effectively prevent both the speculative execution and any associated address translation of memory operations.

3.6 Effect of Different Page Attributes

In addition to fences, page attributes can also influence how speculative memory operations interact with the address-translation machinery. In this subsection, we examine how specific page attributes affect the initiation of page table walks and the caching of the target of a transiently executed memory operation. In particular, we focus on three page table flags: `dirty`, `writable`, and `accessed`.

We leverage our custom kernel module to manipulate the page attributes of the target address. While most page table attributes can be modified without interfering with normal

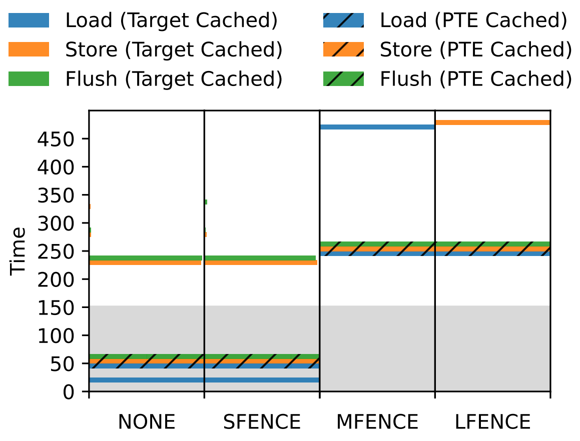

Figure 4: A speculative memory operation (i.e., a load, a store, or a flush) is positioned immediately behind a fence (i.e., SFENCE, MFENCE, LFENCE) inside a transient window. The plot shows the loading times of the target's and its level 1 PTE's cache line. A small loading time, inside the gray rectangle, indicates that the cache line was cached as a side-effect of the transient memory operation. The width of the horizontal bars indicates the frequency of the associated timing value. The results pertain to the Intel i7-8700.

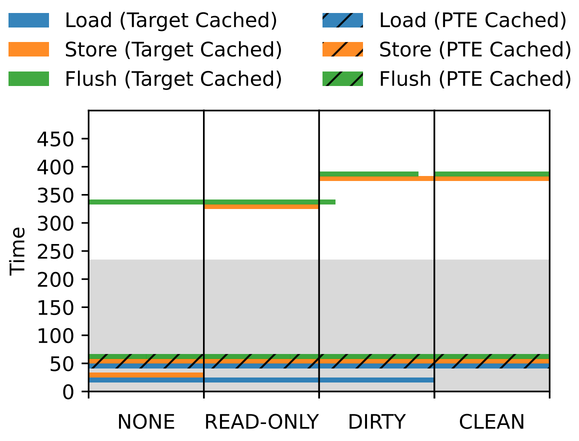

Figure 5: Loading times of a target and its level 1 PTE after a speculative memory operation with a memory operand on a page with the specified page attributes. The tests were run on the Ryzen 5950X.

execution, some attributes might cause exceptions or be automatically modified upon access. Therefore, we set these critical attributes immediately before invoking the RSB gadget, ensuring that only the transient memory operation observes them. After speculative execution of the target, we restore the PTEs to an architecturally safe state. For example, the `accessed` flag must be cleared before every test run, since even the measurement code would otherwise set it again. Note that this restoration step involves a TLB invalidation, which prevents a direct examination of the TLB contents after the transient phase.

We systematically test speculative memory operations targeting pages that are clean (i.e., `dirty`=0), read-only (`writable`=0), or not accessed (`accessed`=0). Across all tested configurations, except possibly Gracemont, where the PTEs are not cached, we observe that a page table walk is always initiated.[5] Furthermore, the address translation is cached except on Summit Ridge, where speculative TLB updates do not appear. Speculative stores to read-only or clean pages trigger page table walks but do not cache the target line (cf. Figure 5). For clean pages, we attribute this behavior to the fact that updating the `dirty` flag is handled by microcode assists that appear to be disabled or not performed during speculation. A similar explanation applies to speculative loads to pages with the accessed bit unset— these loads do not appear to be cached, presumably because the microcode responsible for updating the accessed bit is also not executed under speculation. It should be noted, however, that caching during such accesses cannot be entirely excluded, as the latency of the microcode assist that updates the page flags may exceed the time required for a memory load, even when all page table entries are already cached in the L1 cache.

[5]Compare Table 5 in the Appendix.

Observation 5: Page attributes never prevent the address translation process from being initiated, but they can prevent the target of a speculative memory operation from being cached.

3.7 Effect of Coherency Protocol States

Multiprocessor systems maintain cache coherence by associating each cache line with a coherency state. This subsection examines whether the coherence state of a cache line affects its caching behavior when accessed transiently during speculative execution.

Contrary to the previous experiments, all tests in this section are implemented entirely in user space using multiple helper threads, without involving the kernel module. This setup does not allow us to directly examine potential effects of coherence states on the PTW or TLB. However, we assume that coherence states do not influence address translation, as they are associated with the physical address of a cache line rather than its virtual address.

Depending on the coherence state being tested, one or two helper threads are spawned and pinned to distinct CPU cores other than the test core. In particular, testing the shared state requires the presence of at least 3 hardware cores. Next, the cache line is flushed and the thread conducting the test signals the helper threads: For the exclusive state, a single helper thread loads the cache line, while for the modified state it

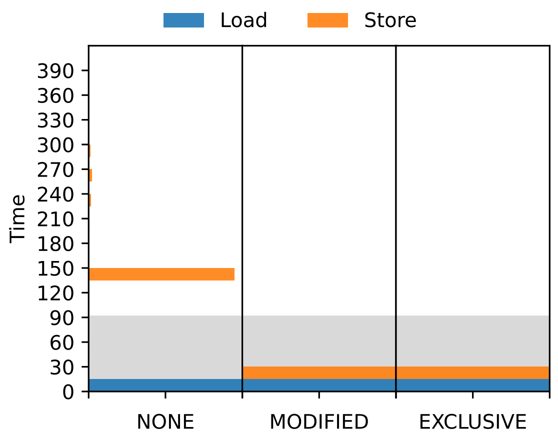

Figure 6: Speculative loads and stores to cache lines either flushed, or in the modified or exclusive coherency states. These results are presented for the Intel i5-6260U (Skylake), where speculative stores do not alter the cache.[7]

stores a value to the cache line. In the case of the shared state, both helper threads load the cache line. After this preparation, the RSB gadget from before is used to trigger a transient window inside which the memory operation is executed.

Interestingly, on all CPU architectures where the memory operand is cached, the memory operand is cached again regardless of the coherency state. Interestingly, on older CPU architectures, where speculative stores to flushed targets do not affect the cache, we now observe that they do when the cache line is in the modified or exclusive state. We speculate that this is caused by the other core holding the cache line directly forwarding it to the requesting core.

Observation 6: Regardless of the coherency state a cache line is in, speculative loads and stores to it can be cached.

4 Covert Channel

The experiments in the previous section showed that speculative stores can influence the data cache on certain processor generations, while on others they only initiate the address translation process without affecting the cache state. For processors where speculative stores modify the data cache, building a covert channel follows the same principle as prior cache-based channels that use speculative loads to exfiltrate data from protected regions. In contrast, on processors where speculative stores do not affect the cache, we construct a covert channel that leverages the page table walk triggered by a speculative store, whose address translation is not cached, to encode information into the caching of page table structures. This demonstrate that the microarchitectural traces left by the page table walks of transient store instructions are sufficiently pronounced to be reliably used to breach process isolation. As the Skylake i5-6260U is the newest Intel processor generation where speculative stores do not affect the cache, we focus on evaluating the constructed covert channel on it.

[7]Our current implementation of the test of the coherency state SHARED requires at least 3 physical cores. Hence this test is not available for the Intel i5-6260U.

Threat Model. We assume two unprivileged user-space processes, a *sender* and a *receiver*, executing on separate cores with distinct virtual address spaces. The sender encodes secret information through speculative stores, while the receiver decodes it by monitoring the cache using Prime+Probe [26]. The receiver is assumed to have access to huge pages. Importantly, both processes operate entirely in user space and are not subject to kernel-level isolation or privilege boundaries, unlike the kernel module used in the previous experiments.

4.1 Sender

The fundamental mechanism relies on the fact that speculative stores trigger page table walks that access level 1 PTEs, which are then observable through cache side channels. When a speculative store is triggered whose translation is not cached in the TLB, the processor performs a page table walk to resolve the address translation. During this walk, the level 1 PTE is loaded into the cache hierarchy, creating measurable cache effects that the receiver can detect.

Setting up the Address Pool. To ensure that page table walks reliably occur and that the accessed PTEs map to LLC sets monitored by the receiver, the sender must exercise precise control over the physical addresses of the level 1 PTEs. However, as an unprivileged user-space process, the sender has limited control over physical memory allocation. The sender addresses this constraint through careful selection of virtual addresses.

Concretely, the sender allocates pages with `mmap` at virtual addresses whose 21 least-significant bits are zero. For 4 KB pages, the virtual-address fields map as follows: bits [11:0] are the page offset and bits [20:12] are the level-1 (PT) index. By setting the bits [20:0] of the virtual address to zero, the sender ensures that the corresponding level 1 PTE resides at index 0 within the level 1 page table. This placement guarantees that the PTE is located at offset zero within the page containing the level 1 page table, which means the least significant 12 bits of the PTE's physical address are also zero. While the sender cannot control the higher-order bits of the physical address (which are determined by the kernel's page allocation), this approach constrains the physical address to page-aligned boundaries. When speculative stores to such addresses trigger page table walks, the resulting level 1 PTE loads are cached, even though the speculative stores themselves do not modify the cache state. By having the receiver monitor a sufficiently large subset of LLC sets that could contain these page-aligned addresses, the sender can reliably induce detectable cache conflicts.

Manipulating the Address Translation. Unlike the kernel module discussed in Section 3.2, the sender cannot execute privileged instructions such as INVLPG to invalidate cached address translations and force page table walks.[8] To ensure that speculative stores consistently trigger page table walks, the sender must prevent the relevant address translations from remaining cached in the TLB. We achieve this by maintaining a pool of virtual addresses and continuously issuing speculative stores across this pool. As speculative stores affect the TLB, stores to different addresses within the pool cause mutual TLB evictions, preventing stable caching of translations. On older microarchitectures (e.g., Intel Sandy Bridge), this behavior takes place automatically, as TLB sets are indexed linearly by portions of the virtual address [14]. Therefore, the pool size must be sufficiently large to ensure that translations are evicted before they can be reused, guaranteeing that each speculative store triggers a fresh page table walk. Below, we provide the actual number of addresses used by the sender and receiver for optimized channel performance.

Pruning the Address Pool. Ideally the receiver only needs to monitor one LLC set, where all of the level 1 PTEs in the address pool map to this set. If the sender is permitted to issue non-speculative loads with a target in its pool of addresses, it can prune its pool to a collection of addresses that form a so-called TLB-PTE eviction set: The level 1 PTEs of the addresses collide in the TLB *and* in the LLC. As long as this eviction set is larger than the largest associativity of the TLB and LLC, issuing speculative loads, stores, and flushes to these addresses in a loop will cause a PTW to be triggered for each as they all contend for the same TLB set and the resulting load of the level 1 PTE will generate significant contention on 1 LLC set, namely that of the target. If, furthermore, these addresses do not themselves contend for the same cache line, repeated speculative loads or even flushes to these address are essentially equivalent to a regular load of the level 1 PTE, which then forms an LLC eviction set.

Concretely, pruning works as follows: ① The receiver is started first and only observes one LLC set. Note that the Prime step of Prime+Probe will naturally evict all other cache lines from the LLC set, including any level 1 PTEs of the senders address pool that map to the LLC set. Note that this explicitly requires that the caches are inclusive. ② The sender first prunes the address pool to addresses that map to the same TLB set. This can be achieved by either building a TLB eviction set or by using the reverse-engineering of TLB hash functions [14] as we elected to do here. ③ To check whether the level 1 PTE of an address in the pool collides with the LLC set selected and monitored by the receiver, the sender first loads the address. Then it uses loads to other addresses in its pool to evict the TLB entry that its initial load created. By offsetting these addresses inside their respective pages it is possible to prevent these other loads from evicting the target of the pool address under investigation from the cache. Finally, the sender loads the original address again. If the load is fast, this means that the target and the level 1 PTE (the TLB entry was evicted) are still cached. Hence the level 1 PTE entry cannot lie in the LLC set monitored by the receiver as, in that case, the Prime step would have evicted it. If, however, the load takes longer, it is likely that the level 1 PTE was evicted as the previous memory operations were especially crafted to avoid the target.

Signal Encoding. Similar to Zhang et al. [36], we adopt the Manchester encoding to transmit information over the channel. The sender encodes information by modulating the rate of speculative store operations. Furthermore, to improve the timing of the RSB gadget, the sender uses an LFENCE instruction instead of CPUID, which prevents subsequent page table walks from starting and reduces the execution time of the gadget from roughly 1100 to 220 cycles. To transmit a high signal (logical '1'), the sender continuously issues speculative stores, inducing frequent page table walks and PTE loads. To transmit a low signal (logical '0'), the sender does not issue store operations, allowing the receiver's eviction sets to remain unchanged in the cache. The actual bit sequence to be transmitted is encoded using Manchester encoding [11], where each bit transition introduces a lower-frequency component in the signal. This ensures reliable synchronization between the sender and receiver.

4.2 Receiver

The receiver's objective is to detect the sender's activity by observing cache effects created when level 1 PTEs are loaded during page table walks. As the sender modulates the rate of speculative stores, the frequency of these PTE loads varies accordingly. The receiver therefore monitors last-level cache (LLC) sets that could contain page-aligned physical addresses of level 1 PTEs and extracts a Manchester-encoded bitstream from the resulting activity trace.

Monitoring the LLC. The receiver observes the covert channel by detecting cache conflicts caused by the sender's speculative stores. Recall that the sender transmits information by inducing page table walks that load level-1 page table entries at page-aligned physical addresses, which in turn create conflicts in the LLC. It then uses a standard *Prime+Probe* technique [21, 26] to monitor these eviction sets. In particular, first huge pages are used to construct a pool of virtual

[8]Even if the sender could use it, INVLPG is likely undesired because it flushes page walker caches, causing additional loads for all levels of page tables. These additional loads increase latency and reduce channel capacity.

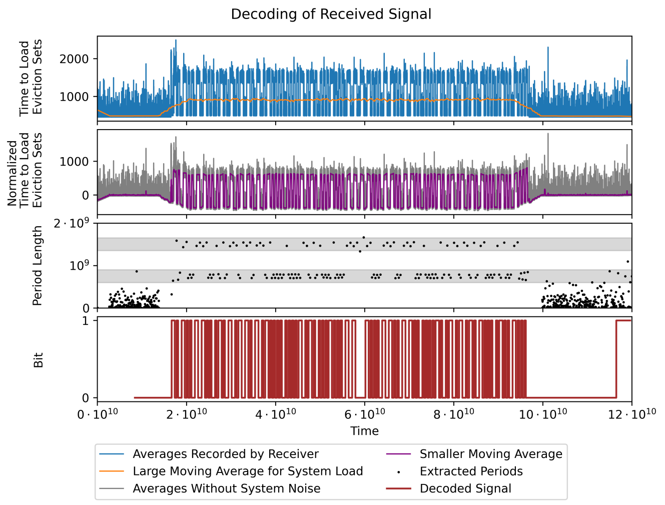

Figure 7: Decoding a noise trace captured by the receiver.

addresses that map to the same set in each LLC slice. This collection of addresses is then pruned to a minimal eviction set of each LLC set by selecting an address in it at random and then iteratively removing an address from the set and testing whether the remaining addresses still conflict with the chosen address. If they do the removed address does not form part of an eviction set of the LLC set that the chosen address maps to.

Decoding the Signal. Given the Manchester-encoded signal, decoding proceeds in three steps, as shown in Figure 7:

1. **Low-frequency noise removal:** To eliminate background noise, such as other processes accessing memory, a moving average with a large window is subtracted from the recorded trace. The window is chosen relative to the sender's Manchester encoding period (e.g., 1024 times the encoding period). Since Manchester encoding ensures equal duration in high and low states, the average will not be perturbed by the sender's signal.
2. **High-frequency noise removal:** To remove fast, transient noise, a second moving average with a much smaller window than the sender's period is applied. This smooths out short-term fluctuations without affecting the encoded signal.
3. **Bit extraction:** We measure the durations of segments where the signal is positive or negative. Under Manchester encoding, the signal flips sign either every sender period or every two sender periods.Accordingly, we keep only segments whose durations fall within tolerances around these expected lengths. A segment of approximately one sender period is decoded as one bit of the sign of smaller moving average; a segment of approximately two sender periods is decoded as two bits with the same sign.

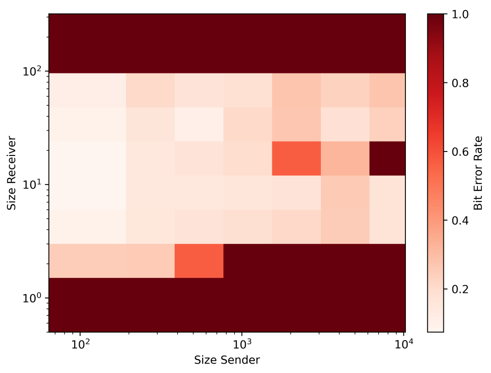

Figure 8: Bit error rate for different sizes of pools of sender and receiver address collections.

4.3 Channel Capacity and Accuracy

Sender's and Receiver's Pool Sizes. We first study the transmission rate and bit error rate as a function of the number of LLC sets monitored by the receiver and the number of addresses in the sender's pool. For this experiment, the receiver collects approximately 500 samples per sender period. The results are shown in Figure 8. When the sender and receiver pools are too small, the monitored LLC sets are unlikely to contain the level-1 PTEs loaded by the sender, resulting in a weak signal. Conversely, when the receiver monitors too many LLC sets, the captured signal becomes too noisy to reliably detect the additional pressure caused by the sender transmitting a logical '1'. We find that monitoring 16 LLC sets and using 128 sender addresses yields a transmission rate of 53 bit/s with a bit error rate of 7%, measured as the average edit distance between transmitted and received data.

Sender's Frequency. We next increase the transmission rate by reducing the number of samples the receiver collects per sender period to below 500. The optimal sender period is 5 120 000 clock cycles, allowing the receiver to collect approximately 160 samples per period. Under this configuration, the channel achieves a bit error rate of 3% while transmitting 175 bit/s.

Address Pruning. When the sender additionally prunes its address pool, the transmission rate can be further increased. In this configuration, we achieve a transmission rate of 346 bit/s with a bit error rate of 13%, using a pruned address set of 120 addresses.

Finally, we evaluate an alternative implementation of the same covert channel by replacing speculative store instructions with CLFLUSH. Using the optimized configuration with address pruning, this variant achieves a bit error rate of 9% at the same transmission rate of 346 bit/s.

4.4 Discussion

The exclusive use of *speculative* stores during the transmission phase provides several practical advantages for the covert channel. As speculative stores can yield exceptions without aborting program execution, the target addresses of these stores need not reside in present pages. Instead, it suffices for the corresponding level-2 page table entry (i.e., the PMD entry) to be present. This significantly simplifies the selection of suitable virtual addresses within an existing process's address space, as it avoids the need to allocate new mappings via `mmap` system calls. More generally, the effectiveness of the covert channel does not depend on the specific memory operation used, but rather on the operation initiating virtual address translations for its targets. Consequently, other memory operations that trigger address translation, such as loads or cache flushes, can also be employed to construct the channel.

It is unlikely that a real-world Spectre gadget directly implements the semantics of a sender with address pruning. However, the existence of gadgets that realize the unpruned variant of the channel remains an open question. A potential real-world gadget for transmitting a secret bit stored in a register could be structured as follows. Depending on the secret value, the gadget issues a load, store, or cache flush to an address chosen from one of two virtual address regions, denoted *A* and *B*, subject to the following constraints: ① All addresses in region *A* have a valid level-2 page table entry (PMD).[9] ② No address in region *B* has a valid PMD entry. ③ Bits [20:15] of the virtual address are zero. ④ The gadget can be coerced into accessing different pages within the selected region. Systematically identifying such gadgets in real-world binaries and assessing their exploitability constitutes an interesting avenue for future work.

Currently, the covert channel transmits its secret through the shared inclusive LLC cache. As it is primarily the burden of the receiver to detect the cache activity of the sender, we speculate that it is possible to extend this covert channel to non-inclusive caches by using a more advanced cache attack, following the approaches shown in [28, 33].

5 Evaluating Speculative Load Hardening against Speculative Stores

One of the widely accepted defenses against Spectre-v1 is to use compiler-based mitigations such as Speculative Load Hardening (SLH) to prevent speculative execution of loads. The idea is to maintain a dedicated flag that reflects the current speculation state and apply that flag to mask (harden) sensitive values so they cannot be exposed. Previous work [38] has explored the efficacy of SLH against Spectre-v1 and propose a stronger variant of SLH, called Ultimate SLH (USLH).

To reduce the performance overhead, conventional SLH hardens only those load operations that have control-flow dependencies on conditional branches whose outcomes may be transiently mispredicted. For such loads, SLH propagates a speculation predicate, derived from the controlling branch, and uses it to mitigate speculative leakage either by hardening the load address or by masking the loaded value. Address hardening ensures that, under mis-speculation, the memory access is redirected to a benign, non-secret location, while value hardening masks the loaded value on mis-speculation, preventing speculative data from propagating through registers or influencing subsequent computations. By default, SLH primarily relies on value hardening and does not universally harden load addresses. In particular, if a secret value is already resident in a register and is used directly to compute a speculative memory address, the corresponding load address may remain unprotected and can still induce secret-dependent leakage. This behavior can be overridden using the compiler flag `--x86-slh-post-load=false`, which enforces address hardening for all loads, except those that access fixed offsets of global and local variables. Moreover, as Zhang et al. [38] note, SLH does not harden speculative stores, which fall outside its load-centric threat model.

In this section, we evaluate the efficacy of SLH and USLH against speculative stores and the PTW-based covert channel introduced in the previous section. It is important to note that these mitigations are typically applied to native C/C++ code. Therefore, we first implement our speculative gadgets (using both RSB and conditional branches) in native C/C++ code. While this is not strictly necessary, it allows us to verify that our results concerning speculative memory operations apply equally well to transient windows caused by conditional and indirect branches. We perform three sets of experiments. First, we compile the gadgets using a pre-built Clang 16 compiler with no SLH flags set. Second, we enable all SLH options that employ speculation predicates as in Table 2. Similarly, we enable all USLH options that rely on speculation predicates except for hardening variable timing instructions (cf. Table 2). Note that for SLH, the non-default flag `--x86-slh-post-load=false` ensures that even the address of the first load after the start of the transient window is masked and for USLH the flag `--x86-slh-store` masks the addresses of speculatively executed stores.

The gadget utilizing RSB prediction follows the similar structure as shown in Listing 2, with a few modifications. Specifically, the return stack must be popped twice, as compilers typically push both the return address and the frame pointer onto the stack (cf. Listing 5 in the Appendix). When SLH or USLH is enabled, we further observe that the compiler pushes two additional words onto the stack, which are used by the callee to track the speculative state. Consequently, the calling function must pop these additional entries to correctly realign the stack pointer during the speculative return.

For the conditional branch predictor gadget, we repeatedly invoke a function (e.g., `gadget_store` or `gadget_load`)

[9]A present PTE is not required.

SLH	`--x86-speculative-load-hardening`
	`--x86-slh-post-load=false`
	`--x86-slh-ip`
	`--x86-slh-loads`
	`--x86-slh-indirect`
USLH	`--x86-speculative-load-hardening`
	`--x86-slh-post-load=false`
	`--x86-slh-ip`
	`--x86-slh-loads`
	`--x86-slh-indirect`
	`--x86-slh-sbhAll`
	`--x86-slh-fixed`
	`--x86-slh-store`

Table 2: Compiler flags for SLH configurations.

within a loop (cf. Listing 4 in the Appendix). This function performs a speculative store (load) to `target` when the control variable `trigger` is true. During the final loop iteration, `trigger` becomes false, but due to prior branch history, the conditional branch predictor incorrectly predicts the branch as taken. As a result, the body of `gadget_store` is transiently executed despite the architectural condition being false. The duration of this transient window can be extended by flushing the `trigger` variable prior to the last iteration, delaying its resolution.

Similar to the conditional branch predictor, the gadget exploiting indirect branch prediction iterates over an array of function pointers (cf. Listing 3 in the Appendix). The last entry of the array is a nop while all other entries point to a function storing the integer value in the target. When iterating through this array, the indirect branch predictor learns the recurring call pattern and speculatively predicts that the last indirect call will also target the store function.

Compiling these speculative gadget variants and measuring the latency of subsequent memory operation (speculative) to the target reveals whether the hardening mechanisms can prevent the speculative execution of the store and load. The results are shown in Figure 9. As expected, SLH prevents the speculative execution of all loads, while speculative stores may still be cached. In contrast, USLH prevents both speculative loads and stores from being reflected in the cache. This behavior is consistent with the respective threat models and security guarantees of SLH and USLH. As the current investigation is conducted entirely in user-space, it is not possible for us to directly test whether a PTW is started on the *virtual address* of the memory operation. In accordance with our investigation in Section 3 it is likely that a PTW will be initiated for the target address. However, this PTW will abort and thus does not leak any information about the actual target address. We further observe that with indirect branch based misprediction neither SLH or USLH stops the load and store from being cached. This is consistent with their threat model not targeting indirect branches as other countermeasures (e.g., retpolines) already exist.

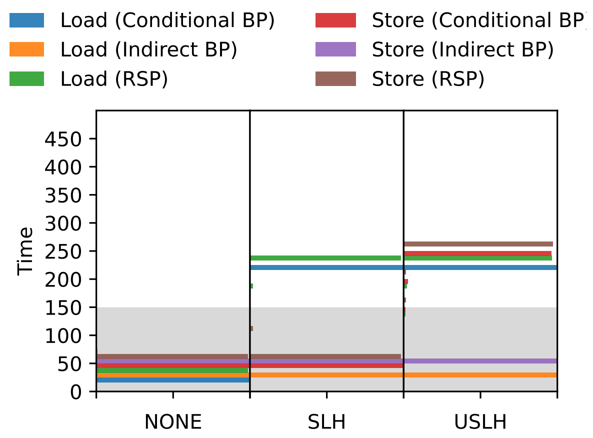

Figure 9: (Ultimate) Speculative Load Hardening applied to gadgets using conditional branches and RSB prediction implemented in native C/C++. The gray rectangle indicates the timing values for which the target are cached.

6 Conclusion

Spectre attacks and their defenses have predominantly focused on speculative loads, leaving the impact of speculative stores and their associated address-translation behavior across different microarchitectures largely unexplored. In this work, we show that speculative memory operations, such as stores and flushes, either update the data cache, similar to load instructions, or initiate page table walks that leave discernible microarchitectural footprint. In particular, we observe that speculative store instructions typically initiate page table walks on all examined Intel and AMD processors. However, the extent of the page table walk varies significantly across microarchitectures. In addition, we investigate the impact of different fence instructions, page table attributes and coherency states of target data on the address translation process and caching behavior of the speculative memory operations.

Based on these observations, we develop a covert channel that leverages the extent of the page table walk to encode information, without relying on store-induced cache fills. Given that the CPUs typically initiate page walks, this channel is generic and works across all processor generations. We further show that popular countermeasures, such as Speculative Load Hardening (SLH), do not protect against speculative store-based leakage, whereas stronger variants such as Ultimate SLH (USLH), can mitigate this leakage.

Acknowledgments

This work has been supported by an ARC Discovery Project number DP210102670 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and through project number 560392681.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE SP*, pages 699–715, 2022.

[2] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *IEEE SP*, pages 1753–1770, 2023.

[3] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos V. Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS*, pages 1046–1060, 2021.

[4] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019.

[5] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck.

[6] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*, pages 769–784, 2019.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019.

[8] Chandler Carruth. Speculative load hardening: A Spectre variant #1 mitigation technique. LLVM Documentation, November 2025. URL https://llvm.org/docs/SpeculativeLoadHardening.html. Last updated 2025-11-05.

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *EuroS&P*, pages 142–157, 2019.

[10] Md Hafizul Islam Chowdhuryy and Fan Yao. Leaking secrets through modern branch predictors in the speculative world. *IEEE Trans. Computers*, 71(9):2059–2072, 2022.

[11] J Kartheek Devineni and Harpreet S Dhillon. Manchester encoding for non-coherent detection of ambient backscatter in time-selective fading. *IEEE Transactions on Vehicular Technology*, 70(5):5109–5114, 2021.

[12] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *ASHES@CCS*, pages 117–126, 2020.

[13] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 2017. URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/aslrcache-practical-cache-attacks-mmu/.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks. In *USENIX Security*, page 955–972, 2018.

[15] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. iLeakage: Browser-based timerless speculative execution attacks on Apple devices. In *CCS*, pages 2038–2052, 2023.

[16] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[17] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, pages 2399–2416, 2021.

[18] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019.

[19] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, August 2018. URL https://www.usenix.org/conference/woot18/presentation/koruyeh.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015.

[22] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient non-observability. In *USENIX Security*, pages 1397–1414, 2021.

[23] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018.

[24] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[25] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box CPUs against speculation contracts. In *ASPLOS*, pages 226–239, 2022.

[26] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, pages 1–20, 2006.

[27] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with secure compilers. In *CCS*, pages 445–461, 2021.

[28] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, pages 2906–2920, 2021.

[29] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[30] Alan Wang, Boru Chen, Yingchen Wang, Christopher Fletcher, Daniel Genkin, David Kohlbrenner, and Riccardo Paccagnella. Peek-a-walk: Leaking secrets via page walk side channels. In *IEEE SP*, pages 23–23, May 2025. doi: 10.1109/SP61157.2025.00023.

[31] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *USENIX Security*, pages 3825–3842, 2022.

[32] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, pages 428–441, 2018.

[33] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019.

[34] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014.

[35] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data. In *MICRO*, pages 954–968, 2019.

[36] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security*, pages 7267–7284, 2023. URL https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-ruiyi.

[37] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution Trojans. In *ASPLOS*, pages 667–682, 2020.

[38] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: Taking speculative load hardening to the next level. In *USENIX Security*, pages 7125–7142, 2023.

[39] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Binoculars: Contention-based side-channel attacks exploiting the page walker. In *USENIX Security*, pages 699–716, 2022.

A Appendix

Microarchitecture	Processor Generation	Processor
Sandy Bridge	**Sandy Bridge**	i5-2400
	Skylake	i5-6260U
Skylake	**Coffee Lake**	i7-8700
	Comet Lake	i7-10710U
Cypress Cove	**Rocket Lake**	i7-11700KF
Golden Cove	**Alder Lake (P-Core)**	i7-1260P
Gracemont	**Alder Lake (E-Core)**	i7-1260P
Zen	**Summit Ridge**	Ryzen Threadripper 1900X
Zen 3	**Vermeer**	Ryzen 7 5950X

Table 3: Tested processors and their microarchitectures.

```
int ret = 0; // Used to prevent the compiler from optimizing out certain loads
uint64_t run() {
    *****trigger = NUMBER_OF_TRAINING_RUNS - 1;

    //  Prepare the array jump_functions.
    for (int i = 0; i < NUMBER_OF_TRAINING_RUNS; ++i)
        jump_functions[i] = store_5_in_target;
    jump_functions[0] = nop;

    for (;;) {
      asm volatile(
          "cpuid\n\t"
          // Flush linked list and target
          "cpuid\n\t"
          : : [target] "r"(target), [trigger] "r"(trigger)
          : "rax", "rbx", "rcx", "rdx");

      // When trigger == 0, this will speculatively store_5_in_target instead of the nop.
      ret += jump_functions[*****trigger]();
      // As this happens after the mis-speculated branch, the conditional branch predictor cannot detect the change of trigger via the branch history buffer.
      if (!(*****trigger))
        break;
      --*****trigger;
    }
  }
```

Listing 3: Speculative Gadget using Indirect Branch Prediction.

```

int gadget_load(uint64_t *target, uint64_t *trigger) {
  int ret = 0;
  if (*trigger) ret = *target;
  return ret;
}
void gadget_store(uint64_t *target, uint64_t *trigger) {
  if (*trigger)
    *target = 5; // This is the speculative store
}

*trigger = NUMBER_OF_TRAINING_RUNS; // Quite a bit of training is necessary to ensure that the branch predictors don't "learn" from multiple runs.
for (;;) {
  asm volatile(
    "cpuid\n\t"
    "clflush (%[trigger])\n\t" // Flush for larger transient window
    "clflush (%[target])\n\t"  // Flush to see the effect of a speculative store
    "cpuid\n\t" // Barrier to ensure that the flushes complete
    :
    : [target] "r"(target), [trigger] "r"(trigger)
    : "rax", "rbx", "rcx", "rdx");
  if constexpr (access_type == AccessType::STORE)
    gadget_store(target, trigger);
  else
    ret += gadget_load(target, trigger);
    // As this happens after the mis-speculative branch, the conditional branch predictor cannot detect the change of trigger via the branch history buffer.
  if (!(*trigger))
    break;
  --*trigger;
}
```

Listing 4: Speculative Gadget using Conditional Branch Prediction.

```
void mispredict_return(void) {
  asm volatile(
    "pop %%rdi\n\t" // Pop entry into mispredict_return (caused by call)
    "pop %%rdi\n\t" // Pop entry of frame pointer rbp that was pushed when entering transient_store
    "clflush (%%rsp)\n\t" // Force speculation
#if (defined SLH) || (defined USLH)
    "add $0x10, %%rsp\n\t" // SLH requires more space on the return stack
#endif
    "lfence\n\t"
    :
    :
    : "cc", "rdi");
}
void transient_store() {
  mispredict_return();

  // This should only ever execute transiently
  *target = 5;
}
```

Listing 5: Speculative Gadget using Return Stack Prediction.

Speculative Loads

Processor	Flush RSP					Invalidate TLB					Flush to PTE1					Flush to PTE2					Flush to PTE3					Flush to PTE4				
	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4
Sandy Bridge	0%	0%	5%	0%	100%	0%	0%	2%	2%	100%	0%	2%	2%	98%	100%	0%	0%	1%	100%	100%	3%	0%	3%	100%	100%	0%	0%	32%	100%	100%
Skylake	0%	0%	1%	1%	100%	0%	0%	0%	100%	100%	0%	0%	0%	99%	100%	0%	6%	96%	99%	100%	99%	97%	98%	99%	100%	99%	98%	99%	100%	100%
Coffee Lake	0%	0%	0%	3%	100%	0%	0%	0%	98%	100%	0%	0%	100%	100%	100%	0%	83%	100%	100%	100%	0%	100%	100%	100%	100%	98%	100%	100%	100%	100%
Comet Lake	0%	0%	0%	0%	100%	0%	0%	0%	100%	100%	0%	0%	93%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Rocket Lake	0%	0%	0%	7%	100%	0%	0%	1%	97%	100%	0%	0%	96%	100%	100%	0%	100%	100%	100%	100%	99%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Alder Lake (P-Core)	0%	1%	3%	6%	100%	0%	1%	3%	9%	100%	3%	3%	14%	94%	100%	4%	6%	92%	94%	100%	24%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Alder Lake (E-Core)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Summit Ridge	0%	100%	100%	100%	100%	7%	100%	100%	100%	100%	9%	100%	100%	100%	100%	98%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Vermeer	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	12%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Speculative Stores

Processor	Flush RSP					Invalidate TLB					Flush to PTE1					Flush to PTE2					Flush to PTE3					Flush to PTE4				
	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4
Sandy Bridge	0%	0%	0%	2%	100%	0%	0%	0%	0%	100%	0%	2%	2%	98%	100%	0%	0%	0%	100%	100%	0%	2%	4%	100%	100%	0%	0%	31%	100%	100%
Skylake	0%	0%	0%	100%	100%	0%	0%	0%	1%	100%	0%	4%	0%	100%	100%	0%	11%	90%	97%	100%	0%	89%	90%	91%	100%	0%	100%	100%	100%	100%
Coffee Lake	0%	0%	0%	98%	100%	0%	0%	0%	3%	100%	0%	0%	15%	100%	100%	0%	82%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Comet Lake	0%	0%	0%	100%	100%	0%	0%	0%	0%	100%	0%	0%	92%	100%	100%	0%	100%	100%	100%	100%	9%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Rocket Lake	0%	0%	1%	98%	100%	0%	0%	6%	100%	100%	0%	0%	96%	100%	100%	1%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Alder Lake (P-Core)	0%	1%	3%	9%	100%	0%	1%	3%	6%	100%	0%	3%	18%	94%	100%	0%	18%	93%	94%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Alder Lake (E-Core)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Summit Ridge	7%	100%	100%	100%	100%	0%	100%	100%	100%	100%	9%	100%	100%	100%	100%	98%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Vermeer	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	5%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Speculative Cache Flushes

Processor	Flush RSP					Invalidate TLB					Flush to PTE1					Flush to PTE2					Flush to PTE3					Flush to PTE4				
	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4	T	PTE1	PTE2	PTE3	PTE4
Sandy Bridge	0%	0%	0%	0%	100%	0%	0%	0%	3%	100%	0%	2%	1%	100%	100%	0%	0%	0%	100%	100%	0%	0%	3%	100%	100%	0%	0%	31%	100%	100%
Skylake	0%	0%	1%	4%	100%	0%	0%	0%	100%	100%	0%	5%	0%	99%	100%	0%	95%	97%	97%	100%	0%	89%	89%	100%	100%	0%	100%	100%	100%	100%
Coffee Lake	0%	0%	0%	1%	100%	0%	0%	0%	98%	100%	0%	0%	100%	100%	100%	0%	83%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Comet Lake	0%	0%	0%	0%	100%	0%	0%	0%	100%	100%	0%	0%	92%	100%	100%	0%	96%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Rocket Lake	0%	0%	0%	7%	100%	0%	0%	4%	99%	100%	0%	0%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Alder Lake (P-Core)	0%	1%	5%	6%	100%	0%	1%	3%	8%	100%	0%	3%	17%	94%	100%	0%	6%	93%	94%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Alder Lake (E-Core)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Summit Ridge	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%
Vermeer	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%	0%	100%	100%	100%	100%

Table 4: Testing whether the page table walk can be interrupted. The transient window is increased from left to right. Entries indicate percentage cached.

Page Permission Bits

Processors		Normal Configuration						Read-Only						Dirty						Accessed					
		Load		Store		Flush		Load		Store		Flush		Load		Store		Flush		Load		Store		Flush	
		T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1
Sandy Bridge	(Sandy Bridge)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Skylake	(Skylake)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Coffee Lake	(Skylake)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Comet Lake	(Skylake)	100%	100%	100%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Rocket Lake	(Cypress Cove)	100%	100%	100%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Alder Lake (P-Core)	(Golden Cove)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Alder Lake (E-Core)	(Gracemont)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Summit Ridge	(Zen)	100%	100%	100%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%
Vermeer	(Zen 3)	100%	100%	100%	100%	0%	100%	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%

Table 5: Different configurations of page flags for speculative stores. Note that the PTE1 loads for the not-accessed page tests are pathological: They stem from setting critical flags.

Processors		No Fence						SFENCE						MFENCE						LFENCE					
		Load		Store		Flush		Load		Store		Flush		Load		Store		Flush		Load		Store		Flush	
		T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1	T	PTE1
Sandy Bridge	(Sandy Bridge)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Skylake	(Skylake)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Coffee Lake	(Skylake)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Comet Lake	(Skylake)	100%	100%	100%	100%	0%	100%	100%	100%	100%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Rocket Lake	(Cypress Cove)	100%	100%	100%	100%	0%	100%	100%	100%	100%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Alder Lake (P-Core)	(Golden Cove)	100%	100%	0%	100%	0%	100%	100%	100%	0%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Alder Lake (E-Core)	(Gracemont)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Summit Ridge	(Zen)	100%	100%	100%	100%	0%	100%	100%	100%	100%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Vermeer	(Zen 3)	100%	100%	100%	100%	0%	100%	100%	100%	100%	100%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Table 6: Transient execution of a memory fence followed by a memory instructions. Entries indicate percentage cached.